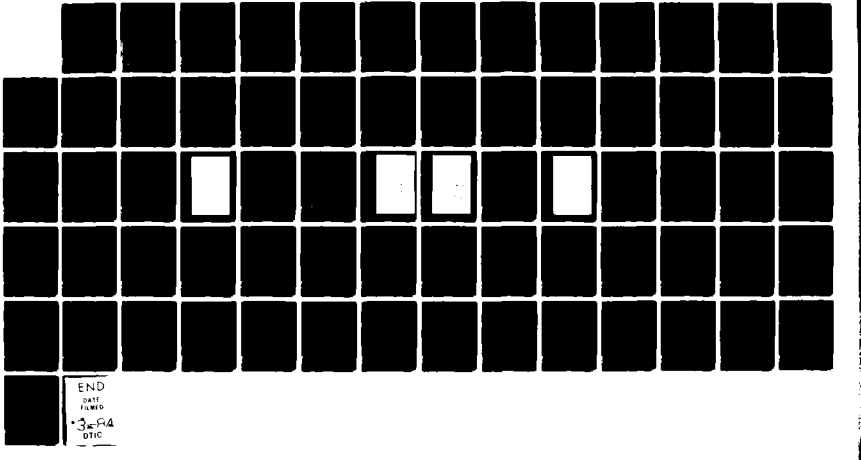
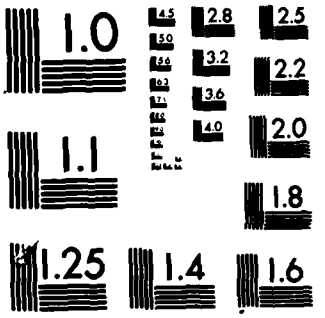
AD-A137 310 A PRELIMINARY EVALUATION OF OBJECT-ORIENTED PROGRAMMING 1/1
FOR GROUND COMBAT MODELING(U) MITRE CORP MCLEAN VA
MITRE C31 DIV R O NUGENT 27 SEP 83 F19628-83-C-0001

UNCLASSIFIED F/G 15/7 NL



END
DATE FILMED
3-84
DTIC




MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A

MITRE

# WORKING PAPER

The MITRE Corporation
MITRE C³I Division
Washington C³I Operations
1820 Dolley Madison Boulevard
McLean, Virginia 22102

WP- 83W00407
No. Vol. Series Rev. Supp. Corr.

CONTROLLED DISTRIBUTION

Subject: A Preliminary Evaluation of Object-Oriented Programming for Ground Combat Modeling

To: Manfred Gale

From: Richard O. Nugent

Contract No.: F19628-83-C-0001

Dept.: W74

Sponsor: AMMO

Date: 27 September 1983

Project No.: 8608A

Approved for MITRE Distribution: Manfred Gale

ABSTRACT: MITRE has implemented the Battlefield Environment Model (BEM) in ROSS, an object-oriented programming language, to provide a demonstration facility and basis for the evaluation of object-oriented programming for the Army hierarchy of combat models. The BEM was constructed in the MITRE Secure Processing Laboratory located in the MITRE Washington offices. This paper discusses object-oriented programming, describes the BEM and provides findings and observations on the implementation of BEM in ROSS and the utility of object-oriented programming.

## ABSTRACT

MITRE has implemented the Battlefield Environment Model (BEM) in ROSS , an object-oriented programming language, to provide a demonstration facility and basis for the evaluation of object-oriented programming for the Army hierarchy of combat models. The BEM was constructed in the MITRE Secure Processing Laboratory located in the MITRE Washington offices. This paper discusses object-oriented programming, describes the BEM and provides findings and observations on the implementation of BEM in ROSS and the utility of object-oriented programming.

## ACKNOWLEDGEMENTS

The author thanks the many people who contributed to this evaluation. In particular thanks are given to Dr. Phil Klahr and Dave MacArthur at the Rand Corporation for providing the computer files, documentation and assistance in bringing up ROSS and SWIRL at MITRE. Within MITRE thanks are given to Bob Conker, Jim Antonisse, Ari Zymelman, John Davidson and Francoise Youssefi for their comments on ROSS and object-oriented programming.

## TABLE OF CONTENTS

## TABLE OF CONTENTS

(Continued)

## TABLE OF CONTENTS

## (Concluded)

## LIST OF ILLUSTRATIONS

## 1.0 INTRODUCTION

### 1.1 Purpose and Scope

As one of four tasks in support of the Army Model Improvement Program (AMIP) Management Office (AMMO) for fiscal year 1983, the MITRE Corporation conducted a preliminary evaluation of the use of object-oriented programming in the AMIP hierarchy of models. In conjunction with this task, MITRE drew upon and enhanced a previous simulation facility to demonstrate and test the application of this programming technique. This paper provides a description of object-oriented languages in general, discusses the simulation constructed with a selected object-oriented language and then presents findings, observations and recommendations regarding the utility of such languages and techniques in the AMIP hierarchy.

### 1.2 Methodology

The methodology for evaluating object-oriented programming was to take an existing model, the MITRE Threat Event Generator, which was previously written in Pascal, and implement it in an object-oriented language. The new model serves as a demonstration facility as well as the basis for the evaluation.

The MITRE Threat Event Generator (MTEG) was a computer simulation model designed to generate the tactical movement, communications and other sensor observables of a Soviet division-sized force at company resolution.[1,2] The list of resulting events provides the basis for the creation of sensor reports which are processed by the ANALYST, an expert system for processing sensor returns to determine enemy activity and critical nodes.[3]

The MTEG was written in Pascal and required enhancement of the threat representation and sensor capabilities in order to provide a more realistic stream of sensor reports to the ANALYST. The combination of the enhanced threat event generator and sensor capabilities became the Battlefield Environment Model (BEM).[4] Figure 1-1 depicts the relationship of the old and new model concepts. The ANALYST, previously written in Pascal, was concurrently rewritten in LISP and operates on a LISP machine.[3]

The language chosen to implement the BEM was the Rule-Oriented Simulation System (ROSS), an object-oriented programming language consisting of a set of LISP macro-instructions and developed at the Rand Corporation specifically for constructing military simulations.[5] The author is not aware of another language or system developed specifically for object-oriented simulations of combat.

Evaluation of the utility of object-oriented programming was thus made by implementing the BEM in ROSS, using the computer facilities of the MITRE Washington Secure Processing Laboratory (SPL). The equipment which supports the BEM includes a VAX 11/780, with two disc drives, a tape drive and a printer, as well as three text and two graphics terminals. Software includes the VMS operating system, EUNICE (a software package to emulate the UNIX operating system), FRANZLISP (the Lisp dialect used to run ROSS) and the ROSS files containing the simulation facilities provided by RAND. The SPL is discussed in more detail in Section 3.4.

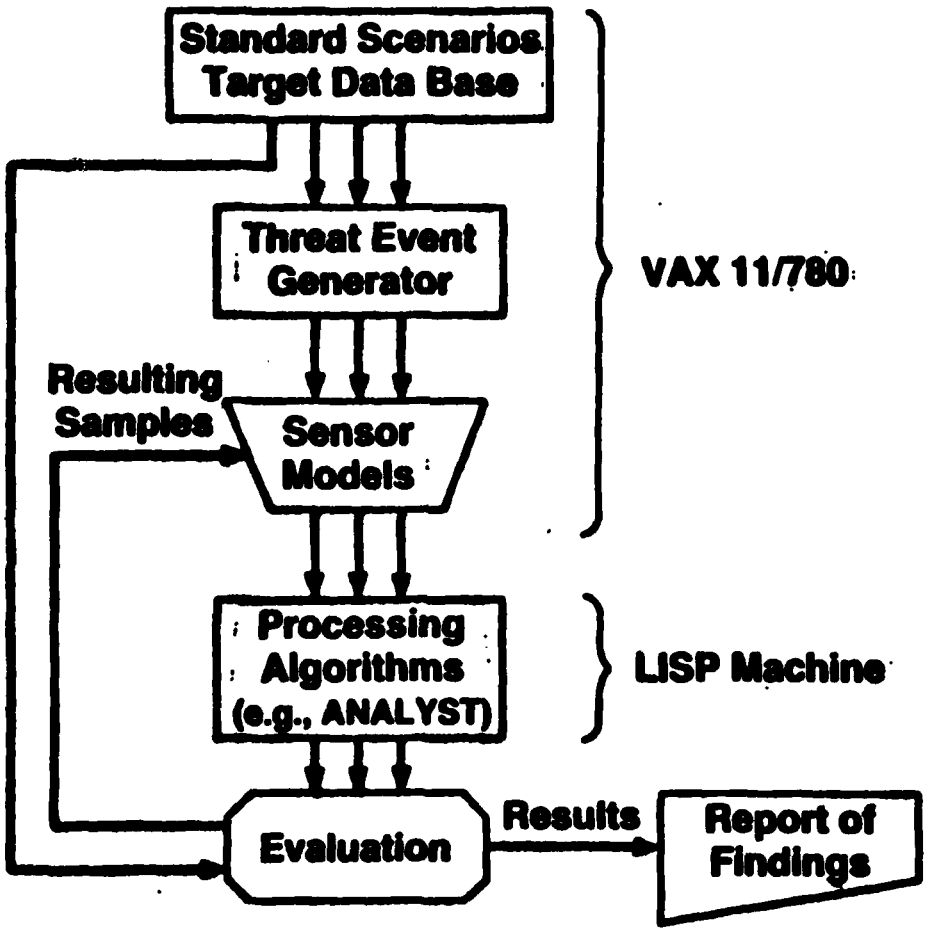



FIGURE 1-1
OLD AND NEW MODEL CONCEPTS

## 1.3 A Brief Description of Object-Oriented Programming

Object-oriented programming is a technique which attempts to overcome the limitations of standard procedural programming. Specifically, as applied to simulation modeling, it attempts to increase the credibility, understandability and traceability of the processes being simulated. In an area in which past models have been weak - that of command and control - object-oriented programming holds forth the promise of lending insight into the $C^2$ processes being modeled. These touted advantages of the technique are the elements examined in our evaluation. The performance criteria for the evaluation are described in the next paragraph.

## 1.4 Evaluation Criteria

Criteria for evaluating the utility of object-oriented programming in Army combat simulations were chosen to provide a basis for decisions on future programming and modeling. The evaluation was conducted with an appreciation of current model limitations, possible applications and resource requirements in terms of computers and programmer/analysts. The evaluation criteria, jointly developed by MITRE and AMMO, and the types of questions addressed are:

- Processing speed. How much is the processing speed of the model affected by the number of objects (units, sensors, etc.) treated in the simulation? What complexity of command relationships and behaviors can be modeled within acceptable running times?
- Processing capabilities. What interface capabilities are there with other languages and operating systems that might be needed to handle complex algorithms or large data bases?
- Graphics interface. What capability does the language have to support interactive graphics displays?

- Ease of model construction, modification and scenario development. How easy or difficult is it to create the necessary objects with their properties and behaviors? How difficult is it to modify units and behaviors to reflect changes in scenario or tactics?

- Traceability. What facilities are provided to trace cause and effect of actions and results?

- Visibility into the Command and Control ($C^2$) process. What credibility and understanding of the simulation does the system provide, especially into the area of command and control?

- Interface with knowledge-based systems. What interface capability does the system provide with knowledge-based systems such as might be required for higher level $C^2$ or intelligence processing?

- Resources. What are the resource requirements in terms of time, equipment, facilities and personnel, training and set-up?

## 1.5 Synopsis of Findings

The following synopsis of findings is presented, addressing the specific criteria as discussed above. A more detailed discussion is given in Section 4.

- Speed. The processing speed of the simulation is affected by both the number of objects and the number and complexity of the object behaviors. While no formula can be easily constructed, it was found that the system could handle 144 Red Units, 8 Blue sensors and 530 auxiliary objects with a simulated-to-CPU time ratio of 2:1, including that required for graphical displays.

- Processing capabilities. The version of ROSS which was used was written in Franzlisp, a LISP dialect which provides convenient interface with other languages.[6] EUNICE, a UNIX software package, was utilized to emulate the UNIX operating system on VMS, the operating system for the VAX 11/780. The capabilities provided by this combination allowed access to the facilities of both operating systems and contributed to ease of development.

- Graphical interface. No direct graphics interface is provided by ROSS. While some artificial intelligence (AI) languages are designed for graphical animation such as dancing figures or turtle-graphics, in ROSS, the interface to graphics devices have

to be built from scratch. However, the object orientation allows the graphics to be conveniently treated as an object to which messages are directed and which draws the desired displays.

- Ease of construction and modification. Basic features of most object-oriented languages, such as object hierarchies and the inheritances of properties and behaviors, aid greatly in the initial model construction as well as in modification. The modularity created by keeping object properties and behaviors together also assists in keeping track of and modifying the simulation.

- Traceability. ROSS offers a number of features which are highly desirable in tracing cause and effect. The message passing characteristic of object-oriented programming facilitates implementing this capability.

- Visibility into the $C^2$ process. Many features, of ROSS specifically and object-oriented programming generally, aid in the intelligibility and understanding of the causes and effects, particularly in the case of command and control. Because of the emphasis on message passing, the dependence of activity on message receipt and reliance of command and control on communication, actions are easily tied to behavioral rules which are initiated by messages received.

- Interface with knowledge-based systems. Interface between the BEM and ANALYST was achieved through the VMS operating system inter-process communications written in C and LISP. Interface on other systems would be dependent on the capability of the underlying language and particular operating system to interface with other languages.

- Resources. Although the physical requirements depend somewhat on the complexity of the simulation, the power and flexibility of the system demand a price of size. Considerable processing size and speed is required to handle all but the simplest of simulations. Personnel training requirements vary depending on the individual background. Analyst/programmers with a LISP background can adapt readily to object-oriented programming, whereas those who are more accustomed to standard programming need time to learn and "think" LISP before tackling ROSS and its use in simulation. Although ROSS itself has relatively few statements to learn and is easy to get a feel for, a working knowledge of LISP is considered essential to fully understand and utilize ROSS or object-oriented programming.

## 1.6 Report Overview

This report continues with four sections. Section Two discusses object-oriented programming in general and presents a description of ROSS. Section Three describes the BEM and its development. Section Four presents findings and observations on implementing the BEM in an object-oriented language. Section Five concludes with a summary and preliminary recommendations as to the role of object-oriented languages in the AMIP program.

## 2.0 OBJECT-ORIENTED PROGRAMMING

### 2.1 Concepts

Several features of object-oriented simulation programming distinguish it from standard procedural language programming. Centermost to object-oriented programming is the passing of messages between objects. All activity is controlled by messages and the resultant behavior brought about by the objects in response to the messages.

Objects may represent components of the system being investigated, for example, the individual military units of a force. Knowledge regarding each object, that is, its attributes and behaviors, is stored with the object rather than in separate data arrays and procedural subroutines. The resultant modularization makes it easy to trace cause and effect and to make modifications. The behaviors of objects are IF-THEN rules which describe the actions to be taken by an object if a particular message pattern is received.

Another feature of object-oriented programming languages is the object hierarchy and the ability of objects to inherit attributes and properties from their parents in the hierarchy. The hierarchy assists in the construction of the simulation as well as in sending messages. Whenever an object receives a message whose pattern is not stored directly with the object, a search is made of parents in the hierarchy to find the indicated behavior. Properties, such as speed, location, etc., are similarly treated. This feature cuts down on the amount of storage required and shortens the time needed to construct the simulation. In compiled code, accessing inherited behaviors and properties does not involve significant losses in processing speed, however interpreted code can slow down during these operations.

Object-oriented languages are written in, or built upon, LISP or LISP-like languages. LISP is the basic language (in this country) for artificial intelligence. It provides for symbolic manipulation and list processing, and is more amenable to dealing with decision-making procedures and rule-based behavior. A desirable feature attained with the use of LISP is the ability to operate in the interpreted mode for the development and testing of the simulation.

## 2.2 ROSS

### 2.2.1 Background

ROSS was developed by the Rand Corporation to conduct military simulations taking advantage of advances in artificial intelligence and computer development. ROSS is based upon LISP and has been written in the LISP dialects Maclisp, Franzlisp and Interlisp. It consists of a set of LISP macros and allows ROSS and LISP functions to be interspersed in the programming.

ROSS was applied in a simulation of air combat* and was successful in providing a testbed for investigating various tactics and employment techniques. This evaluation looked at its applicability to ground combat simulation.

### 2.2.2 Description of ROSS Basic Features

The basic features of ROSS are the actors, with their properties and behaviors, and the messages which invoke the behaviors by the actors. Actors are created and caused to interact through ROSS commands.

---

*Simulating Warfare in the ROSS Language (SWIRL)[7]

2.2.2.1 ROSS Commands. ROSS commands are of the form

(<ask or tell><object><message>).

The parentheses are a LISP characteristic and set off a list, or function. "Ask" and "tell" are equivalent in ROSS and form a LISP macro alerting an object to receive a message. "Object" is any actor in the simulation, and "message" is the message sent to that actor.

2.2.2.2 Actors. Actor is a term used interchangeably with object but which connotes some degree of animation and autonomy. There are two types of actors, basic and auxiliary.

Basic actors are used to represent components of the system being simulated or under study. These are the basic objects of the simulation. To facilitate construction and control of the simulation, there are two types of basic objects: generic objects, representing classes of objects, and instances of the class, called instance objects or instance actors. Instance actors are the physical components or units being represented, for example "tank company #1", whereas a generic actor might be "tank company".

The other type of object is the auxiliary object. An example of an auxiliary object in the BEM is the Mathematician which is used to perform most of the calculations and thereby allows the code associated with the basic objects to be less cluttered. Other auxiliary objects are used to assist in the creation and control of the simulation.

Actors may have properties which represent, for instance, fuel levels, number of vehicles, plans, etc., the data associated with a particular object. Properties may be inherited. Actors may also have behaviors. These

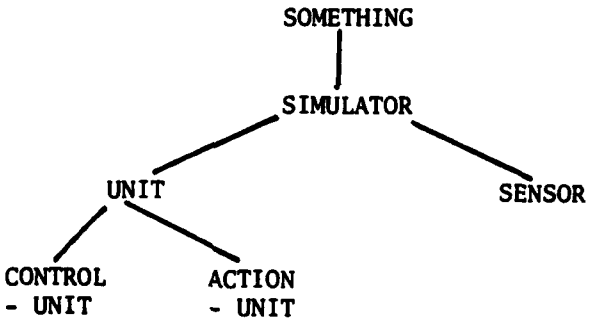
behaviors are written in the form of IF-THEN rules. They list actions to be taken by the actor upon receipt of a message pattern. The message acts as a stimulus and the resultant actions are the actor's response. The response generally includes sending messages to other actors. Actions taken by actors can thus be directly linked to the messages they receive.

2.2.2.3 Object Hierarchy. The parent-offspring relationship implied by the generic-actor (class of objects) and the instance-actor (object instance) is organized into an object hierarchy. Succeeding levels imply super classes, or classes of classes. This linking of objects allows inheritance by an actor of properties and behaviors of generic actors representing the class or classes of objects to which it belongs. The hierarchy does not need to be a strict tree but may be a complex hierarchy allowing an object to have multiple parents, indicating that it might be a member of more than one class. This complex hierarchy permits the inheritance of properties and behaviors from more than one parent. (Care must be taken to prevent conflicting inheritances.) The hierarchy assists in the creation of the simulation by allowing common features to be entered and stored at the highest level. During the simulation, information on an actor may be queried or used in the simulation by doing an inheritance search (i.e., going up through the hierarchy) if the data is not stored at the actor level. In the SWIRL simulation, an example of multiple-inheritances is given of an AWACS aircraft which inherits properties and behaviors from two parents - the generic actors Moving-Object and Radar. Properties and behaviors associated with movement for the AWACS are

inherited from the actor Moving-Object, whereas properties and behaviors for the AWACS associated with radar detection and capabilities are inherited from the actor Radar.

2.2.2.4 Predefined Actors and Primitive Behaviors. Four auxiliary actors exist in the ROSS environment to assist in the creation and operation of the simulation or to handle errors and diagnostics.

Something is the top-most actor in the object hierarchy. All of the primitive ROSS behaviors are stored in Something. One of these behaviors allows the user to use Something to create all of the other actors in the simulation. In the BEM hierarchy, to be discussed in the next section, part of the hierarchy might be represented as follows:




The highest level ROSS actor Something created Simulator when tasked with this code

(ask something create generic simulator)

In two separate messages Simulator was then asked to create generic actors Unit and Sensor. The actual code is:

> (ask simulator create generic actor unit with . . . <the properties Unit has>).

In turn Unit was asked to create generic actors Control-Unit and Action-Unit. These steps were possible as each actor inherits the primitive "create" behavior from the top-level, Something.

Nclock is another special actor which represents the simulation clock and controls the time stepping of the simulation.

ROSS-error is a special actor which provides error-handling in the simulation. The other auxiliary actor is Property which prompts the user for property values during actor creation.

2.2.3 Actor Creation and Manipulation

Actors with their properties and behaviors are created before the simulation, utilizing the predefined ROSS actors, their primitive behaviors and the object hierarchy. During the simulation the properties, behaviors, memory (facts known), and plans (things to do) of each object can be manipulated. (Actors could also be created or removed during the simulation, either dynamically or interactively). There are 14 behaviors in the actor Something to create actors, one behavior to kill actors (remove them from the simulation), and four behaviors to manipulate actor behaviors.

2.2.3.1 Creation of Generic Objects. The ROSS command to create generic objects is of the form:

> (ask something <or a descendant in the hierarchy> create generic <generic-object-name> with <properties>).

For example, if Unit had already been created, it could be asked to create Control-Unit:

(ask unit create generic control-unit).

In this case no additional properties are assigned to Control-Unit. ROSS automatically assigns the following properties: type, parents, and descendants, with the appropriate values.

2.2.3.2 Creation of Instance Objects.The ROSS command to create instance objects is of the form

(ask <generic-object> create instance <object> with <properties> ).

If a generic tank company, Tank-Co, had been created, it could be asked to create an instance Tank-Co-1 as follows:

(ask tank-co create instance tank-co-1 with position (10.5 20.3) fuel 50 <etc> ).

In this case Tank-Co-1 is assigned all the attribute values pertaining to its individual status. It would also inherit values such as the number of vehicles from the generic Tank-Co. These values, while initially common to all tank companies, could be varied for each instance during the simulation.

A number of ROSS commands exist to expand the basic creation capability. These commands allow the user to create objects by analogy, recreate the objects or remove them from the simulation.

2.2.3.3 Actor Behaviors. Actor behaviors are defined as a set of actions to be taken on the receipt of a message or class of messages. This relationship is set up by a ROSS command of the following form:

(ask <object> when receiving <message-template> <actions> ).

This behavior definition is in the form of an IF-THEN rule. The LISP macros translate this to: "if the object gets a message matching the message template, it is to take the indicated actions." The use of message templates is made possible through pattern matching and variable symbols. This feature greatly extends the power of message passing and reduces the number of behaviors which have to be defined. Variable symbols in ROSS are prefixed with the character ">" for single variables (LISP atoms) and with "+" for multiple variables (LISP lists).

An example behavior, such as might be in the BEM model, which shows the use of pattern matching, is as follows:

```
(ask sensor when receiving
  (>unit is in your coverage area)
  (~you add !unit to your list of targets-in-range)).
```

In this case a sensor instance will, upon receipt of a message that a particular unit is in range, add that unit to its list of targets in range. The tilde symbol "~" indicates an abbreviation and will be further explained in the discussion on abbreviations below. The "!" indicates that the value of the following variable should be used, not the variable itself. The "!" is an evaluation macro which is necessary because ROSS, unlike most LISPs, is a non-evaluating dialect.

Other ROSS commands exist to

- recall actor behaviors, that is, print the behavior to the terminal, to assist the user in debugging or understanding actions in the simulation, and
- forget or kill actor behaviors, so that the actor will no longer follow the actions indicated in the behavior.

2.2.3.4 Actor Memories and Plans. Actors may also have plans and lists of known information. These, unlike properties and behaviors, cannot be inherited. They can be modified dynamically or interactively during the simulation and can be queried by the user. Plans can be made and unmade. It is this planning feature which works together with the simulation clock in ROSS to provide a pseudo-time stepping of the simulation.

The message to the actor which causes planning to take place is of the form:

(ask <actor>

plan after <n> seconds <action>)

For example a message sent to a unit might be

(ask tank-co-1 plan after 10 seconds send confirmation call)

where "send confirmation call" is another message which would be sent to the unit (!myself) after the 10 seconds to invoke another behavior.

2.2.4 Conduct of the Simulation

The elements of the simulation are the initiation, control of the simulation itself, manipulation of the actors and behaviors, and the reporting, tracing and debugging facilities.

2.2.4.1 Initiation. All activity is controlled by messages, so that any actors that are to be active in the simulation must receive at least one message at the beginning or during the simulation. In the case of military units these messages might simulate actual orders communicated.

2.2.4.2 Control. Control of the simulation is maintained by the simulation clock. The clock maintains a time-ordered list of events in the form of event time and event actor. Upon completion of the current event, the clock is advanced to the time of the next event and then a message is sent to the appropriate actor. Once alerted, the actor looks to its list of things-to-do; the message stored there invokes a behavior and the actor takes the actions indicated.

The simulation is therefore event-oriented and pseudo-time-stepped.

2.2.4.3 Actor Manipulation. Actors properties and behaviors may be dynamically altered in the simulation or, by interrupting the simulation, the user may enter changes and then continue the simulation. Care must be taken in the latter case because of the effects on events which have been scheduled but not executed prior to the time of the interrupt.

2.2.4.4 Tracing. Behaviors with the actor Something provides a facility for tracing actor behaviors. The user asks the actor to trace a particular set of behaviors. Tracing can be done either with or without inheritance and is invoked with ROSS commands of the form:

(ask <actor> trace your behavior matching <message-pattern> .)

For example, it might be desirable to trace movement messages being sent to actor Tank-Co-1. This could be done with the following command:

(ask tank-co-1 trace your behavior matching (+ move +)).

This would result in a trace of all messages sent to tank-co-1 with the word "move" in them. The tracing option is undone by sending a similar type message to the actor of the form:

(ask <actor> untrace your behavior matching <message-pattern> .)

2.2.4.5 Reporting. The user can select the messages to be reported on the screen as the messages are passed, or kept in a file to be printed out for analysis. The recorder functions are behaviors of the actor Something. The three behaviors are:

- to record message transmissions to an actor into a designated file when the message matches a designated pattern
- to unrecord (stop recording) messages as indicated above
- to record every message transmission to an actor into a file

For example, if it is desired to record all messages to Tank-Co-1 in a file tank-msg-file, the following command could be sent:

(ask tank-co-1

record every message transmission into tank-msg-file.)

2.2.4.6 Graphics. Because graphics are typically hardware dependent, there is no specific package provided by ROSS for displaying graphics. ROSS documentation however does indicate the need for graphics in visualizing what is happening in the simulation.

The BEM uses an actor called "Artist" to whom all messages are sent concerning graphics. This use of an auxiliary object to handle graphics allows the code having to do with graphics to be centralized and to free the actor behaviors of code which is neither ROSS nor LISP.

### 2.2.5 Other Facilities

Editing of actors, properties and behaviors can be done using any line or screen editor before the simulation is begun. It is much more powerful and effective, however, if the user can suspend the ROSS environment, make the necessary editing, either temporarily or permanently changing the file with the actor behaviors, and then reenter the ROSS environment. This is possible, for instance, in EMACS running under VMS or UNIX, with the right combination of editors and operating systems.

In the SPL, ROSS is used in conjunction with the EMACS editor; this provides a flexible system which allows the ROSS process to be interrupted, on-line editing to be done, and the process resumed. This can be done using the windows and buffers of EMACS which allow viewing of both processing and editing at the same time.

ROSS is quite readable as it is written. The abbreviation and message-passing conventions tend to be self documenting and thereby assist the user or others in understanding and modifying the code. ROSS provides an abbreviations package which makes the code even more readable. Abbreviations are prefixed with the tilde character, "~". An example is "~your", the abbreviation for "ask myself recall your".

In the example given earlier

```
(ask sensor when receiving ( >unit is in your coverage area)
        (~you add !unit to your list of targets-in-range))
```

the "~ you" is an abbreviation of "ask !myself" where "myself" is a variable indicating the current actor.

## 2.3 Other Object-Oriented Languages

There are a limited number of language systems which have been applied to object oriented programming in the past. Some of these are no longer used or are used now in only limited applications. Several additional language systems have been developed recently or are being developed at this time. The list provided is not all-inclusive especially for some of the more recently developed languages.

### 2.3.1 SIMULA

SIMULA is an early object-oriented language which introduced communications between objects and incorporated an object hierarchy but no inheritance of properties or behaviors. It is ALGOL-based.[8]

### 2.3.2 ACTORS

ACTORS is an early object-oriented language which explored parallelism. It was developed by Carl Hewitt at MIT. It is sometimes referred to as being in the ACT class of object-oriented languages.[9]

### 2.3.3 SMALLTALK

SMALLTALK is a language designed to assist users in communicating and interacting with personal computers. It was developed in 1972 by Alan Kay at the Xerox Palo Alto Research Center (PARC). It utilizes message passing between objects and incorporates inheritance of behaviors and properties through an object hierarchy.[10]

### 2.3.4 DIRECTOR

DIRECTOR is an object-oriented language especially designed for graphics animation and artificial intelligence applications. It was designed and implemented by Kenneth Kahn at MIT.[11]

### 2.3.5 LOGO

LOGO is a LISP-based micro-computer language used to introduce programming and develop problem-solving skills. It was developed at the MIT Laboratory for Computer Science and is best known for its application to turtle graphics. LOGO employs message passing generally from the user to objects such as the "turtle" on the graphics terminal.[12]

### 2.3.6 FLAVORS

FLAVORS incorporates message passing and lattice inheritance whereby objects can inherit properties and behaviors other than from parents. FLAVORS underlies the powerful window editors/processors of the new LISP machines.

FLAVORS was developed at the Artificial Intelligence Laboratory at Massachusetts Institute of Technology. It is a language feature of ZETALISP, the LISP dialect used on the LISP Machine.[13]

### 2.3.7 LOOPS

LOOPS is a recent object-oriented language which incorporates a rule-based inheritance scheme. It uses four programming paradigms:

- procedure-orientation, the INTERLISP-D base of LOOPS
- object-orientation for the organization of data
- access-orientation for monitoring programs by other programs
- rule-orientation for representing decision-making knowledge

LOOPS was developed at XEROX PARC and is based on INTERLISP-D. It borrows from FLAVORS the multiple-inheritance lattice.[14]

## 3.0 DESCRIPTION OF THE BATTLEFIELD ENVIRONMENT MODEL (BEM)

### 3.1 Model Overview

The BEM consists of the necessary data bases, preprocessors which convert the data into actor files, and the additional actor files constituting the threat and sensors. The purpose of the BEM is to produce a realistic flow of sensor reports to the MITRE ANALYST (a knowledge-based system for fusing sensor data) while allowing interactive control of the Red units, Blue sensors, and their behaviors. The BEM also provides a testbed for investigating the effects of changes in Red behavior and Blue sensor employment. The BEM was implemented in ROSS to provide a basis for the evaluation of the utility of object-oriented programming in ground combat simulations. For a more detail description of the BEM, see the MITRE Technical Report.[4]

### 3.2 Scenario

The simulation builds upon data taken from the Scenario Oriented Recurring Evaluation System (SCORES)[15] for a division-sized threat force. Battalion positions, both at the beginning and end of a selected six-hour time slice were expanded to company level by analysis of the terrain and Soviet unit deployment.[16,17,18] Unit movement takes place over a nodal network which includes the road network and trafficable paths for company-sized units. Figure 3-1 depicts the transportation network for the BEM.

### 3.3 Actor-Objects in the Simulation

Red company-sized units and headquarters up to division level make up the threat side of the objects in the simulation. The friendly side consists of

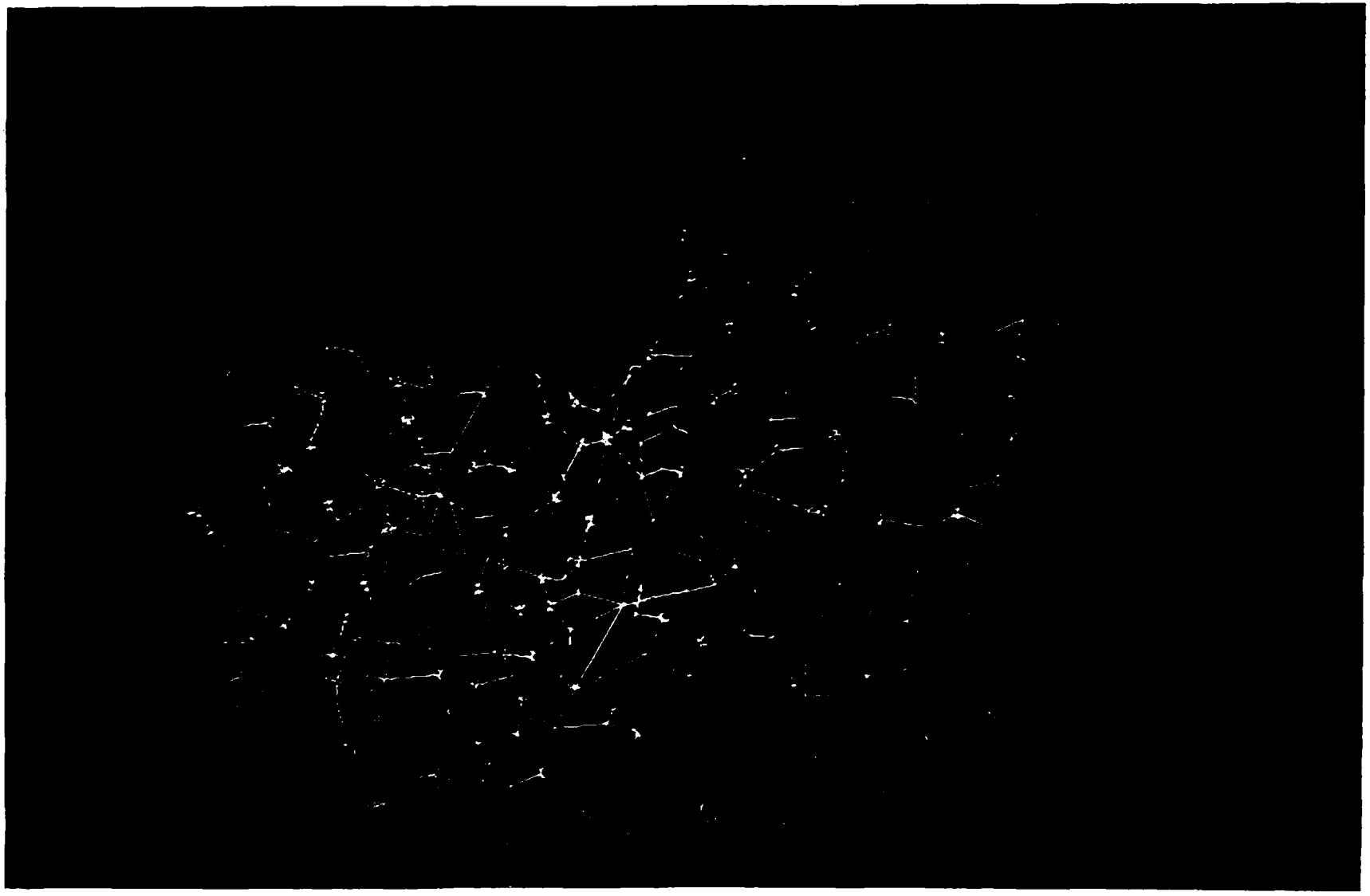


FIGURE 3-1
MOVEMENT NETWORK FOR THE BEM

Blue intelligence, surveillance and target acquisition sensors and their controlling units. Figure 3-2 depicts the object hierarchy for the creation of these basic objects and the auxiliary objects used to control the simulation. Selected actors will be described below in order to give an understanding for the construction and operation of the simulation. A graphical display of the troop positions is given in Figure 3-3. Figure 3-4 displays representative Blue sensors in position and their coverage.

### 3.3.1 The Simulator

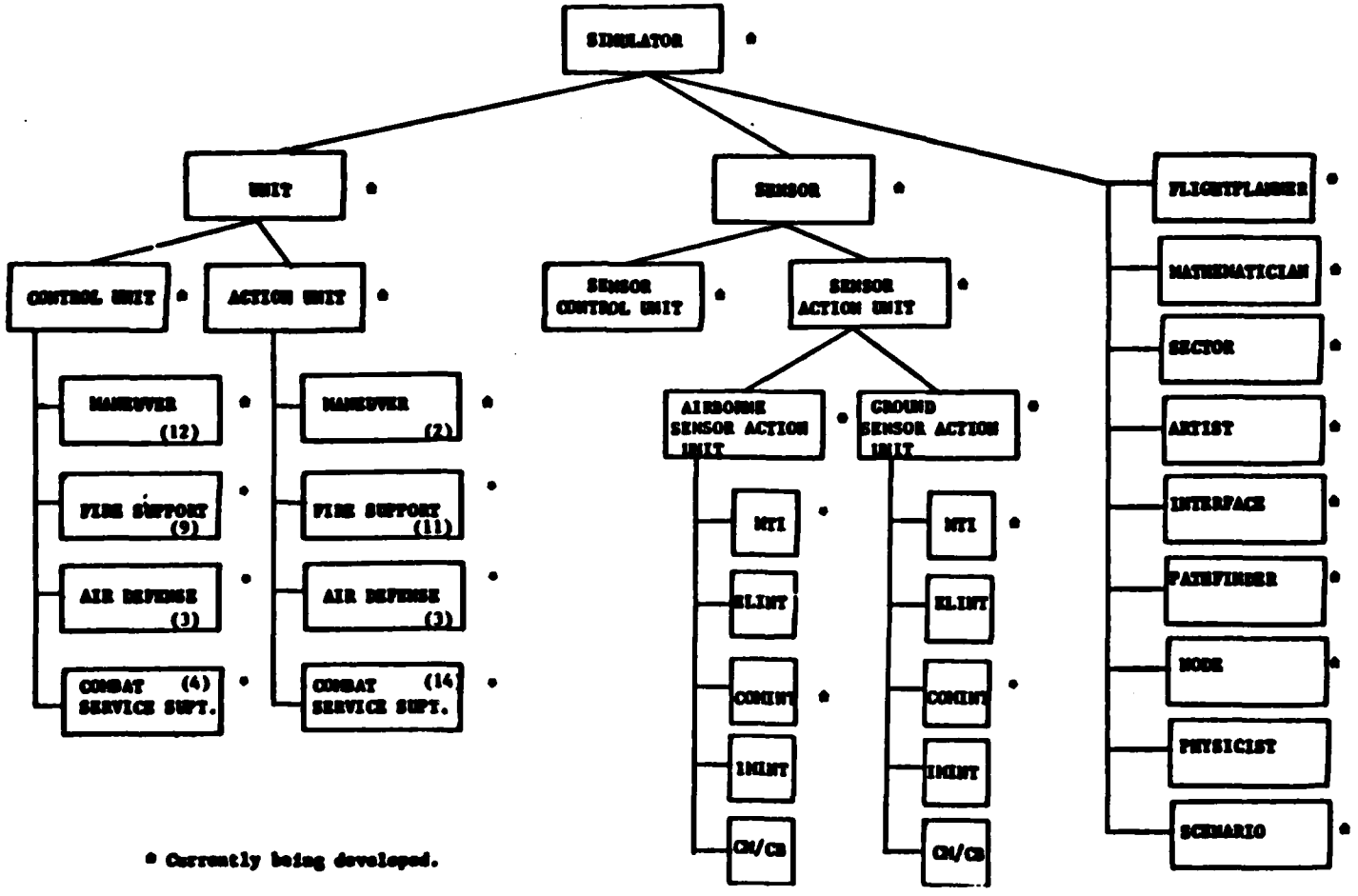
The Simulator is the top-level simulation object used to initiate the creation of all other objects and to control the simulation. The Simulator is created from the ROSS object Something in the BEM, the Simulator is in turn asked to create the Unit and Sensor objects and each of the auxiliary objects whereas other objects are created from their parents.

The properties of the Simulator are default values for the graphics terminals and parameters for initiation, timing and termination of the simulation.

There are four behaviors for the Simulator in addition to those inherent in ROSS. These essentially set-up the scenario data, prepare the graphics display (including drawing the transportation network, terrain and unit positions) and then start and control the speed of the simulation.

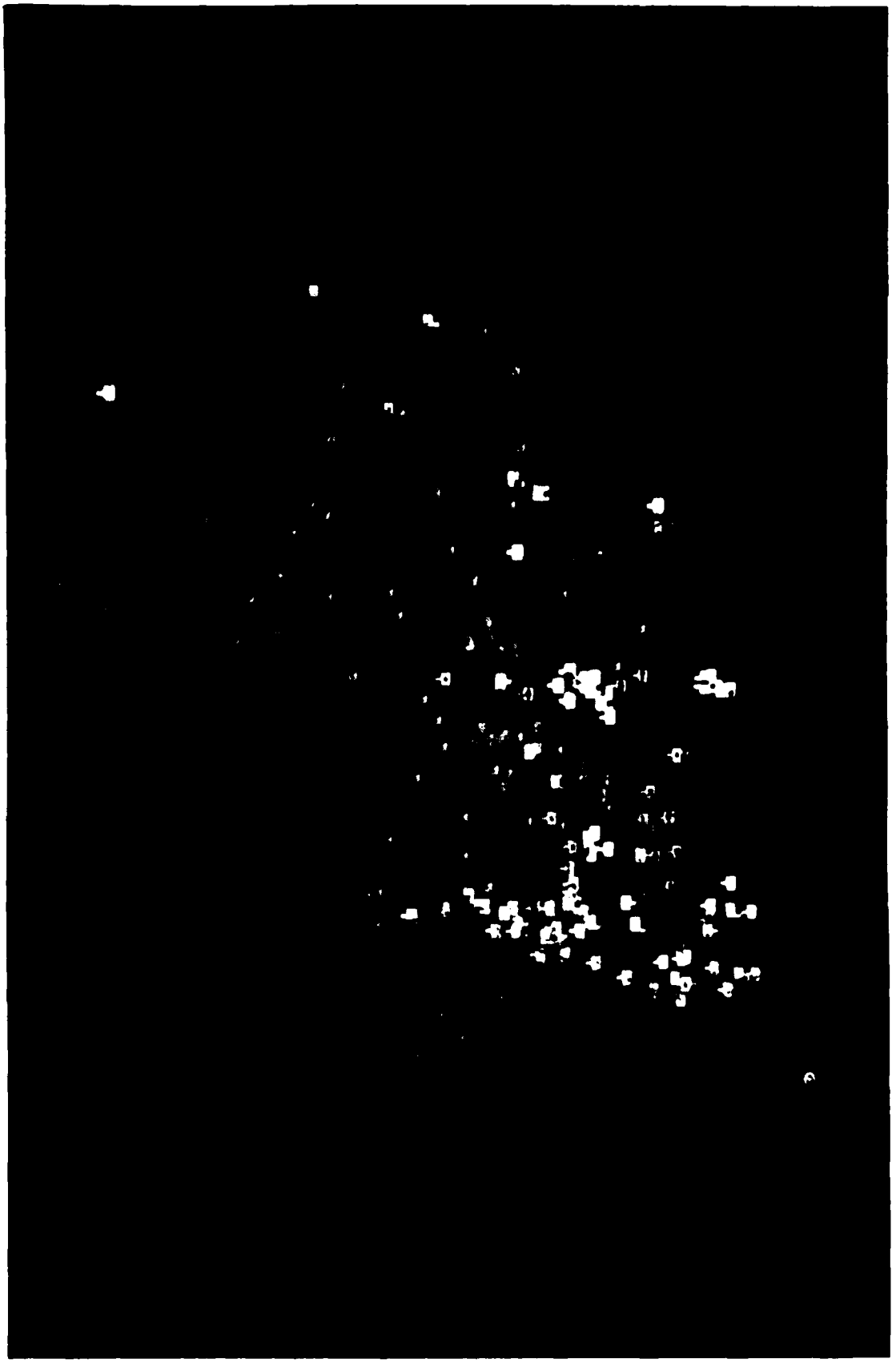
MITRE

## STRUCTURE OF BATTLEFIELD ENVIRONMENT MODEL (BEM)




* Currently being developed.

FIGURE 3-2
OBJECT HIERARCHY IN THE BEM



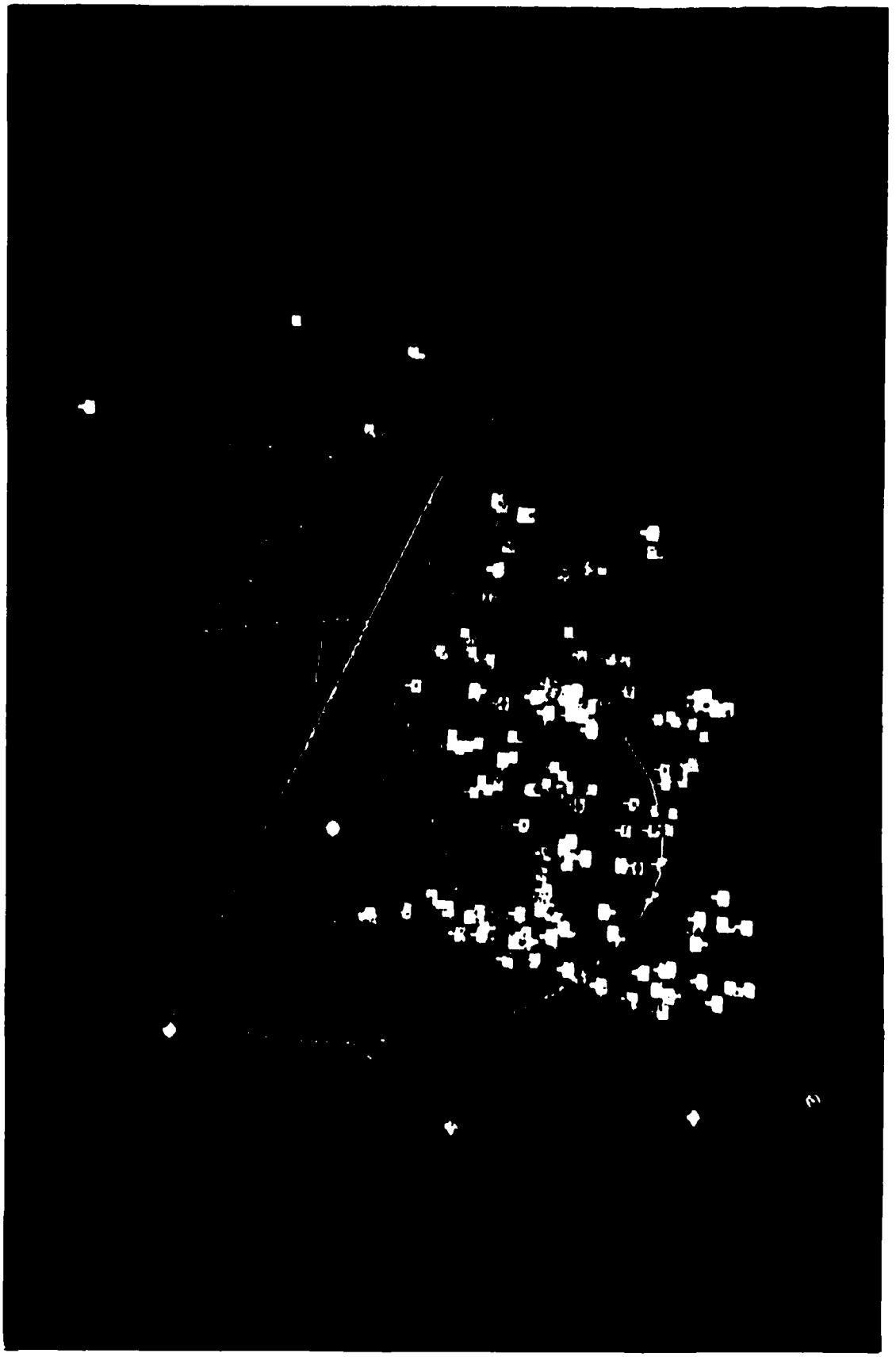


FIGURE 3-4
BLUE SENSOR COVERAGE IN THE BEM

### 3.3.2 The Sector

Sector is a special auxiliary object designed to localize detections by some of the sensors. It also improves the integrity of the simulation by acting as a third party in accessing data. The need to localize detections is caused by

- the event-orientation of the simulation versus the continuity of some activity such as movement and,
- the problem of identifying which sensors, if any, have coverage of an area in which a unit is moving.

Once the match-up of target and possible detecting sensors is made, more detailed calculations can be made as to probabilities of detection and then the appropriate data can be processed through the Sector rather than have the sensor have access to more target data than necessary. Figure 3-5 depicts the individual sectors, which inherit their behaviors from Sector but have their individual property values.

The properties of the individual sectors are

- position, in terms of four x-y locations of a rectangle
- units-list, a list of all units moving in the sector at the time, dynamically changed as units move in and out of the sector, and
- sensors-list, a list of all sensors having coverage in some portion of the sector at the time and requiring target-sensor matching and data access through the sector. (See the discussion of sensors for explanation of which sensors require this matching).

There are several behaviors stored at the Sector and accessible by the individual sectors for the determination of unit enterings or departing and of sensor coverge. An example behavior is that when a sector receives a message that a unit has entered the sector, (this is sent by the unit based on

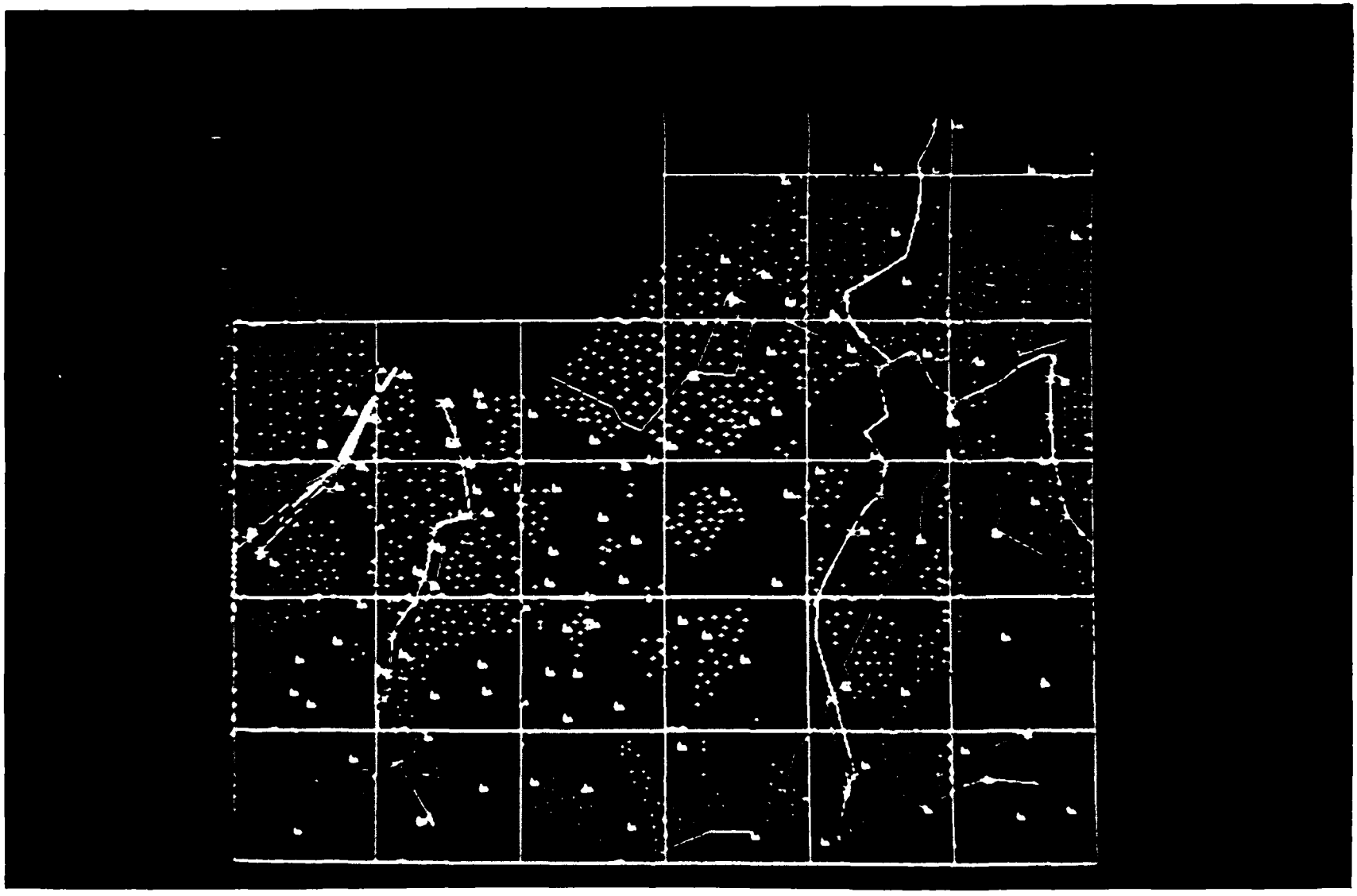


FIGURE 3-5
SECTORS IN THE BEM

its movement plan) the sector adds the unit to its unit-list and plans ahead to remove it from the unit-list based on the unit's calculated departure time. It then sends a message to the Mathematician to determine if there are any time and location overlaps between the unit and any MTI sensor currently having coverage in the sector.

### 3.3.3 The Artist

The Artist is an auxiliary actor whose purpose is to control all interface between the simulation and the graphical display terminals. The Artist has been written to be relatively independent of the particular graphical devices available by calling low-level LISP functions to drive the graphics terminals. Because the behavior of the Artist does not assist in the understanding of the military simulation, the functions the Artist performs are written in *LISP*.

The properties of the Artist include the parameters to set up the real-world coordinates of the simulation.

There are currently 17 behaviors for the Artist which include drawing the transportation network, terrain representation, and unit symbols at their initial locations, indicating movement, communications and artillery firings. An interactive capability was constructed for "picking" an actor using a graphics cursor and displaying its designation and properties. The user can interactively use the Artist by interrupting the simulation and querying units, sensors, sectors and network nodes as to their status and activity. The Artist maps the real-world simulation coordinate system selected (stored as Simulator properties) to the coordinate system utilized by the graphics terminals.

### 3.3.4 The Pathfinder

The Pathfinder is an auxiliary actor that is used both in the preprocessing stage and dynamically during the simulation to determine routes over the transportation network for units and sensors. The Pathfinder accesses unit and network data and determines feasible routes with path-finding algorithms written in LISP.

The Pathfinder does not have any properties but has three behaviors:

- to determine a unit nodal route by time
- to determine a ground sensor nodal route by time
- to move either a sensor or unit over the nodal path at the determined times

### 3.3.5 The Mathematician

The Mathematician is the work horse of the auxiliary objects and performs most of the simulation calculations. The unit behaviors are thus freed of unnecessary and repetitious calculations which do not help in the understanding of the simulation. Many of the functions are written in LISP.

The Mathematician does not have any properties.

The types of behavior for the Mathematician are indicated above.

There are nine behaviors for the Mathematician:

- to calculate object update points in between events
- to determine time and space overlaps between an entering unit and any MTI sensors currently covering a sector
- to reconstruct cycle definition for a sensor
- to determine time and space overlaps between a unit and any COMINT sensor currently covering a sector

- to determine sector penetration of an airborne sensor, i.e., when a sensor starts coverage in any part of a sector
- to determine sector penetration for a unit
- to determine sector penetration of a ground sensor
- to determine coverage by an airborne sensor

### 3.3.6 The Interface

The Interface is an auxiliary object which provides the interface between the main simulation module of the BEM, the ANALYST and the Blue Sensor Control Station (BSCS) (see section 3.4.1). Sensor reports are temporarily stored by the Interface, are accessed by the ANALYST or BSCS during the simulation run, and are then erased by the using facility.

### 3.3.7 The Control-Unit

The Control-Unit is a generic actor which assists in the creation of the subordinate generic control units and stores common control unit behaviors.

The properties of the Control-Unit include the parent (Unit) and the list of subordinate control-units, (e.g., a generic tank battalion headquarters) automatically assigned in ROSS.

The single behavior currently stored at the Control-Unit is that giving the actions to be taken upon receipt of the message "implement battle plan". The resultant actions are for the Control-Unit to

- send messages to the Artist to display an acknowledgement communication
- send messages to subordinates to implement the battle plan
- send a message to the sector in which the unit is located that a communication has occurred
- cause the unit itself to be moved if required

### 3.3.8 The Action-Unit

The Action-Unit is a generic object created to assist in the simulation construction and to store the common action-unit behaviors. The properties of the Action-Unit include the parent (the Unit) and the list of generic subordinates (e.g., a generic tank company), again both handled by ROSS automatically.

The principal behavior of the Action-Unit is that which occurs when it receives the message "implement battle plan". The resultant steps are the same as those for the Control-Unit, except that the Action-Unit does not have subordinate units to notify.

### 3.3.9 The Sensor-Control-Unit

The Sensor-Control-Unit assists in the creation of subordinate ground or airborne sensor control units and contains the common properties and behaviors for these units.

The Sensor-Control-Unit has three designated property slots: target priorities, graphics shape and ground-sensor-cycle. The graphics shape is common to all and is specified at this level.

There are two behaviors for the Sensor-Control-Unit:

- to send a ground-type sensor to a designated point with a specified mission
- to send an airborne-type sensor on a specified search pattern in a particular area

### 3.3.10 The Sensor-Action-Unit

The Sensor-Action-Unit assists in the creation of the subordinate generic airborne or ground sensor action units which in turn create the generic

units representing the types of sensors in the simulation (currently moving target indicators (MTI), communication intelligence (COMINT) sensors, electronic intelligence (ELINT) sensors, counter-mortar/counter-battery (CM/CB) radars, and image intelligence (IMINT) sensors). The Sensor-Action-Unit also contains the behaviors common to all sensor action units.

The properties of the Sensor-Action-Unit include the parent (the Sensor), the list of subordinate generic actors, and parameter slots for the sensor and platform capabilities to be filled in at either the generic or instance level, such as the mission definition (flight path, duration, duty-cycle etc.), current status, range, and radio frequency.

The three behaviors for the Sensor-Action-Unit are:

- to move ground sensor to a designated point with a specified task
- to move an airborne sensor in a racetrack pattern with a specified task and plan
- to fly to a designated point

## 3.4 Demonstration Capability

The demonstration facility for this project is located in the MITRE Washington Secure Processing Laboratory (SPL). Figure 3-6 depicts the physical layout of the SPL. Figure 3-7 shows the hardware and software configuration for preprocessing and operation of the BEM, and interface with the Blue Sensor Control Station and the ANALYST.

### 3.4.1 BEM Work Stations

Preprocessing of data files can be done on any available text terminal using a graphics terminal if desired to visually check unit locations and movement plans.

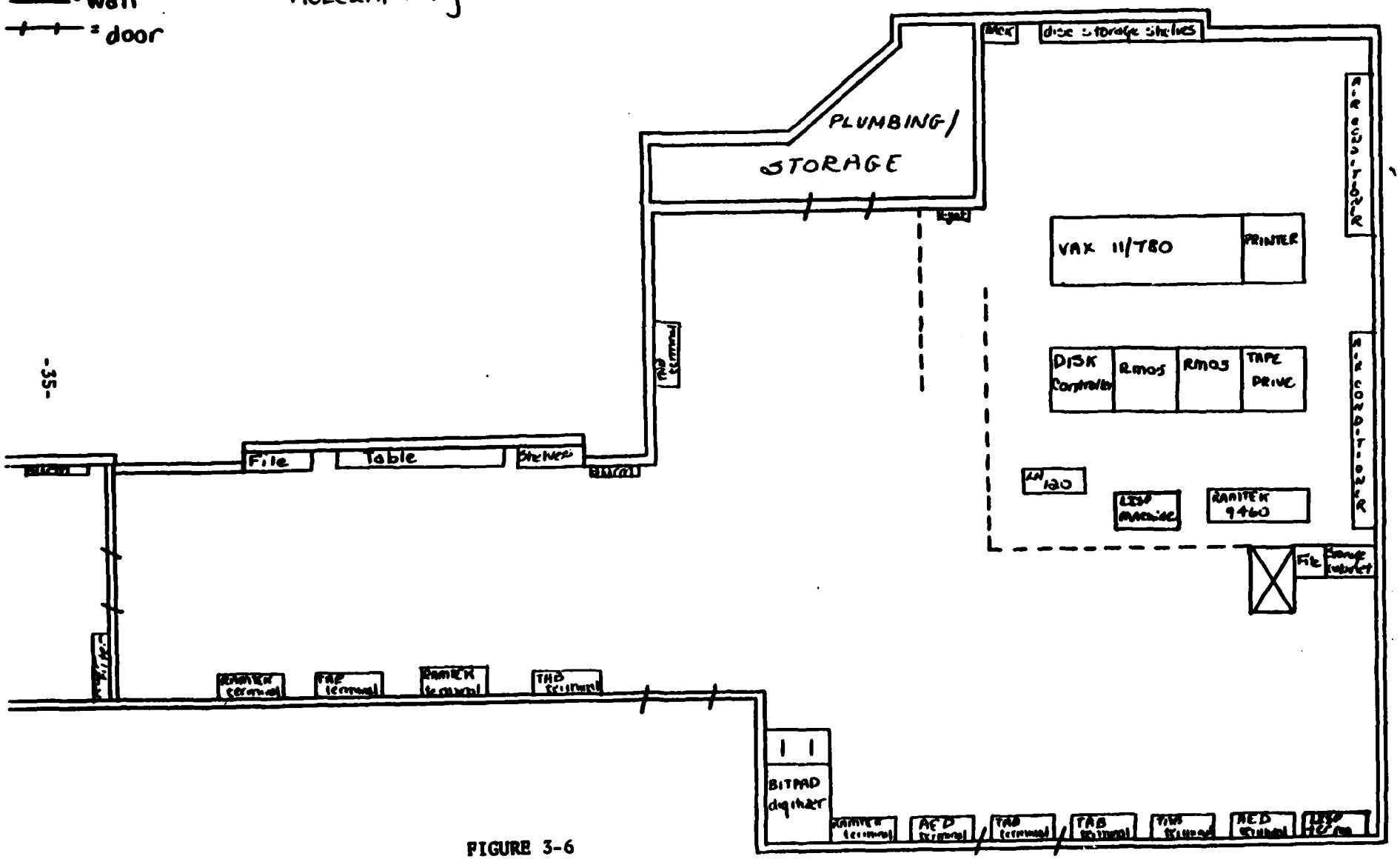



FIGURE 3-6
PHYSICAL LAYOUT OF THE SPL

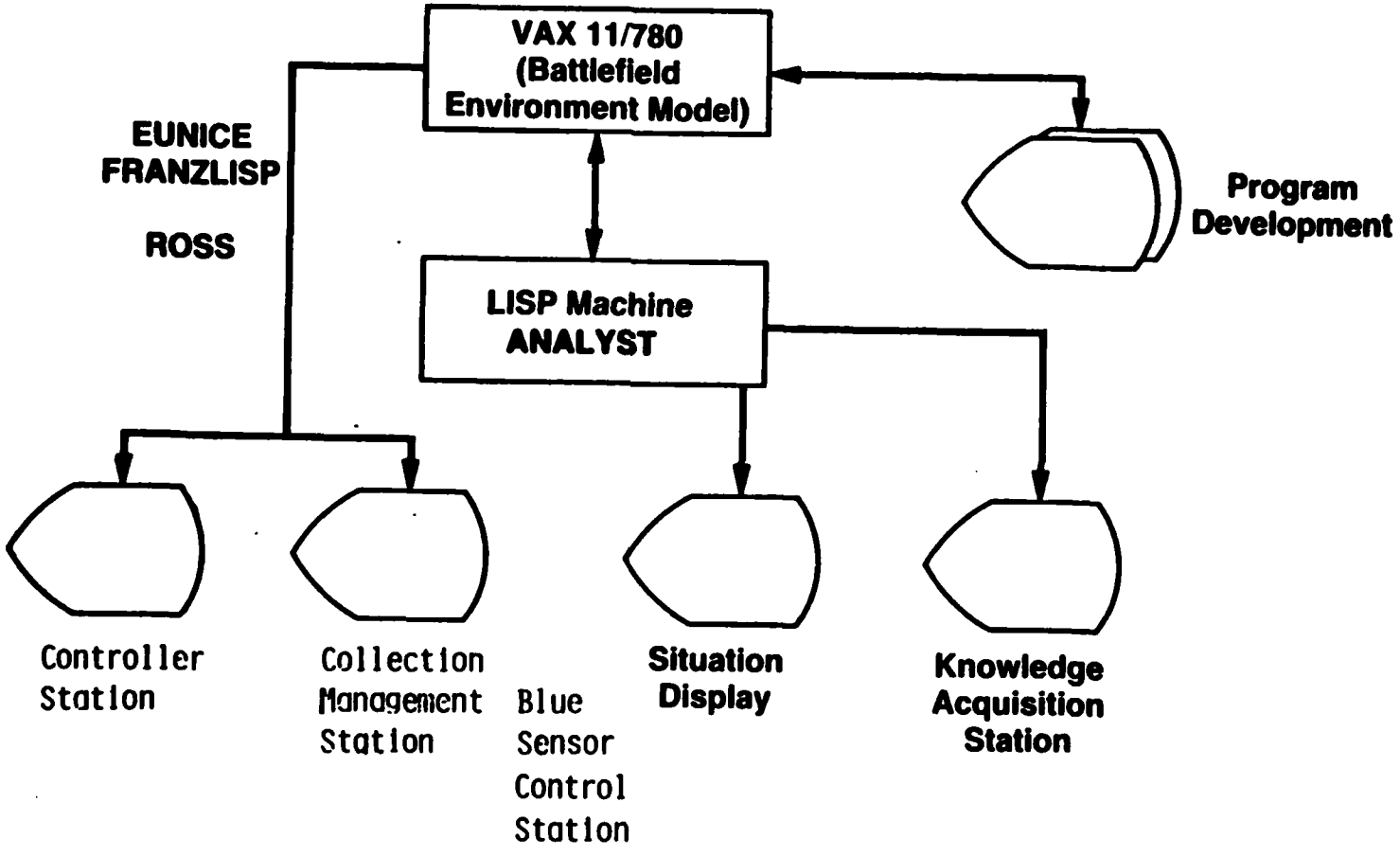



FIGURE 3-7
CONFIGURATION OF THE SPL

Conduct of the simulation requires the operation of the Controller Station consisting of a text terminal and a graphics terminal, and the Blue Sensor Control Station consisting of a graphics terminal.

At the Controller Station the simulation is initiated and changes can be interactively entered from the terminal; the positions and activities of the units and sensors are exhibited on the graphics display. The Blue Sensor Control Station is used to display sensor data and sensor reports as well as to interactively task sensors on a mission basis.

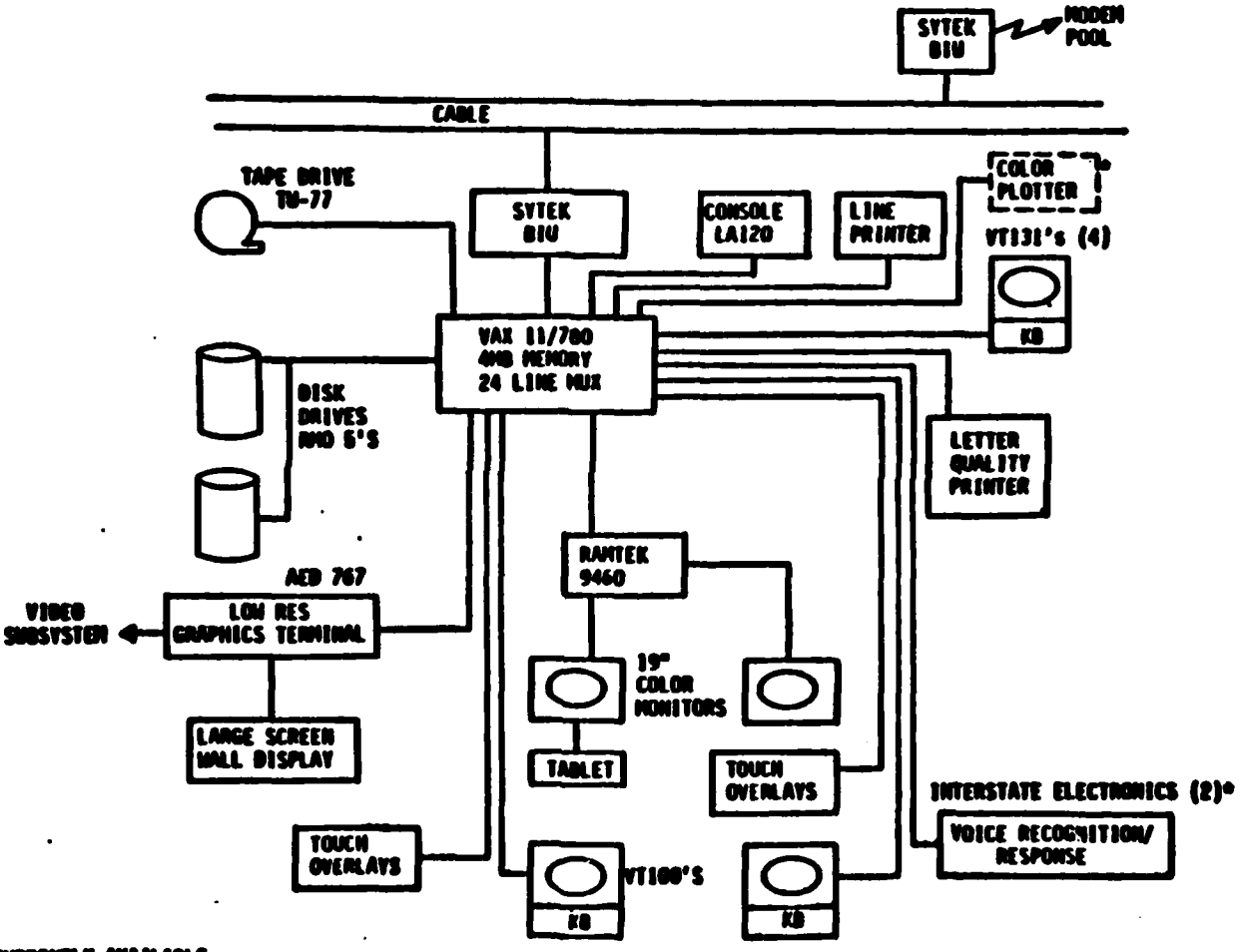
The program development, preprocessing of data, and simulation operation are conducted on terminals operating from a VAX 11/780 with one 67 megabyte disk drive, one 300 megabyte disk drive, and 8 megabytes of memory.

The SPL is located entirely within a shielded vault which is being cleared as a special compartmented information facility (SCIF).

### 3.4.2 Briefing Facility

Located within close vicinity of the SPL is the MITRE Command and Management Information System (CAMIS) Laboratory which contains a similarly configured VAX 11/780. The CAMIS Laboratory also includes a VIP briefing/conference room with projection facilities for showing large screen displays from the graphics terminals. Demonstrations of the BEM are provided in the CAMIS Laboratory by physically transporting the programs and data via computer tapes.

Figure 3-8 presents the computer configuration for the CAMIS Laboratory.




*NOT CURRENTLY AVAILABLE

FIGURE 3-8
CAMIS LABORATORY CONFIGURATION

## 4.0 FINDINGS AND OBSERVATIONS

Our findings follow the list of evaluation criteria given in Section 1.3. Additionally, observations are presented either with the finding where they are related or following the findings. The presentation of each finding is preceded by a short discussion on rationale or desired levels of capability where these are felt to be applicable.

### 4.1 Processing Speed

#### 4.1.1 Desirability

To be useful to the Army at one of the three levels in the AMIP Model hierarchy, a language should be able to create a simulation capable of matching the appropriate combinations of scale, battle length and running time. At the corps/division level the simulation should be able to handle three to five days of battle time with units at company or battalion resolution and approximately a ratio of 20 or 30:1 simulated-to-CPU time. The complexity of the simulated command and control processes should be sufficient to allow the simulation to run uninterrupted for the length of battle.[12,13]

#### 4.1.2 Capability

The BEM Simulation includes 144 Red units, 8 Blue sensor units, and approximately 530 auxiliary objects. The number of auxiliary units could be reduced considerably by storing the network data in arrays rather than having 490 objects representing the nodes. However, the nodes do not have any behavior and so their effect on processing time should be minimal. With these conditions, the BEM is able to attain a simulated-to-CPU time of approximately two-to-one when graphics are run and four-to-one without graphics.

The ROSS documentation discusses the speed problem and indicates that distributed processing may be one way to overcome this deficiency, however preliminary results with simulated parallelism shows the improvement to be less than expected.[21] With any object-oriented programming, the price paid for the power and flexibility is memory size and processing speed. New hardware, that is, machines designed for symbolic processing, may hold promise to overcome these problems.

## 4.2 Processing Capabilities

### 4.2.1 Desirability

It would be desirable to be able to interface the object-oriented language with other languages or programs for a number of possible reasons. One application might be in using the object-oriented language to handle a command and control module as part of a large scale simulation written mainly in procedural languages. Or the simulation could be mainly object oriented while relying on a few modules written in procedural languages to handle complex algorithms or to manipulate large data bases. Another application might be in accessing higher resolution models written in procedural languages.

There is another aspect of processing capabilities which has to do with the capability to interrupt the simulation, suspend operations while editing or querying is done, and continue the simulation at the point interrupted or at any earlier point in the simulation. The capability to interrupt, suspend operations and continue at the point of interrupt is needed to develop the simulation and make modifications. The capability to restart the simulation

at another point would give the user/analyst real power in looking at the effects of changes in tactics and decision making as indicated in the actor behaviors.

#### 4.2.2 Capabilities

The ability to interface with another language is a function of not ROSS but of the LISP version in which ROSS is written. The version of ROSS used was written in Franzlisp. This was run on the VAX by using the EUNICE emulation package wherein the UNIX shell (necessary for running Franzlisp) is run as a VMS (the VAX operating system) job. Franzlisp provides the facility of operating with other languages and the ability to utilize FORTRAN, Pascal, or C functions or subroutines.

Use of the UNIX shell and the EMACs editor provided the capability to interrupt the simulation, perform editing and/or querying, and continue the simulation. The EMACs editor also provided the capability to edit ROSS or LISP in one window of the terminal screen and to see the evaluation done in another window.

### 4.3 Graphical Interface

#### 4.3.1 Desirability

Dynamic graphics are required in a simulation to assist the user throughout the preparation, conduct and analysis of the simulation. During the preparation of the simulation, graphical representation is an aid in the development and verification of data bases for input such as terrain and unit positions. During the simulation, a dynamic representation of the battle events allows the user to follow the conduct so that the simulation may be

stopped and corrected if a logical sequence is not being followed. After the simulation has been run, a playback capability assists the user in tracing cause and effect and analyzing the overall simulation.

#### 4.3.2 Capability

The capability to support interactive graphics is not a function of the simulation language itself but of its capability to link to the operating system, the graphical equipment and the links between the CPU and the graphics terminal. ROSS does not provide any direct capability to support graphics. Object-oriented programming does, however, lend itself to treating the interface with the graphical equipment as an auxiliary actor. Messages are sent to the Artist (or other appropriately named actor) to initiate the graphical terminal, erase the screen, draw terrain background, draw unit positions, etc.

The SWIRL simulation provides an interface with the graphical equipment at Rand. Because of the difficulties mentioned before, this is not generally portable. For the BEM, we used the actor Artist to interface with the graphics terminals. Both the control and collection management work stations were provided with an interactive query capability for assistance in program development and modification.

### 4.4 Ease of Program Construction and Modification

#### 4.4.1 Desirability

The simulation language needs to provide the user/analyst with adequate facilities to construct and modify the simulation to model tactics and decision making. This criterion is related to those of traceability and visibility in that

those capabilities assist in finding the functions and relationships which need to be modified.

#### 4.4.2 Capability

Object-oriented programming results in a modularity of data and coding that makes it easy to locate data and behaviors which require modification. All of the properties and behaviors associated with an actor are stored with the actor or a parent. Use of ROSS requires storage of the data and behavior for each actor in a separate file. This makes it possible to call up the file for the actor, or the appropriate parent, and perform the required changes. It also simplifies the steps required to compile the code.

The English-like nature of ROSS code makes it easy to read and understand the actors behaviors, thereby indicating what editing has to be done to effect the necessary changes.

The ability to use the interpreted mode in LISP (and ROSS) is a distinct advantage in the initial development of a simulation. It is also possible to use a combination of interpreted and compiled code during the later development or modification stage so that the effect of changes can be tested without the need to recompile.

### 4.5 Traceability

#### 4.5.1 Desirability

In order to validate the model and track down errors in logic or coding, it is necessary to have a means of tracing cause and effect of various actions in the simulation. To do this, it may be necessary to trace a sequence of messages and behaviors to determine the relationship of the initiating messages and end result.

### 4.5.2 Capability

The message passing aspect of object-oriented programming makes it easy to trace cause and effect because all actions must be initiated by messages, and the actor behaviors explicitly state the actions which are initiated by each message.

ROSS provides a capability for recording or displaying selected messages for the user to track down selected message types or all of the messages going to particular actors. This allows the user to determine the causes of events by seeing the messages which are passed and, by looking at the behaviors, what actions are taken.

## 4.6 Intelligibility

### 4.6.1 Desirability

One of the major problems with previous models has been the lack of visibility into the command and control processes. Visibility into the $C^2$ process adds to the credibility of the simulation and the analyses to which it contributes.

### 4.6.2 Capability

Intelligibility and particularly visibility into the $C^2$ process are inherent in object-oriented programming because actions can only take place due to message receipt and the behavioral rules for the actors. This parallels the heavy reliance of command and control on communications on the battlefield. The dependence of actions on message receipt allows all action to be traced by tracking the messages which are sent and received.

As stated before, there are several ROSS commands which permit tasking the actors to trace a particular behavior or all behaviors. Other commands exist to cancel the trace commands once they are no longer desired. ROSS also provides the facility to display selected messages on the screen or place them in a file for later reading. This permits watching progress of the simulation either in the form of all messages or in selected messages to help in understanding a portion of the simulation.

We feel that a graphical display of the simulation is invaluable in following and understanding. It is also useful to be able to interrupt the simulation and to interactively query actors about their behaviors, status or activity. While these features are not built into ROSS the inherent data structure of object-oriented programming made it easy for us to implement these features.

## 4.7 *Interface with Knowledge-based Systems*

### 4.7.1 Desirability

While object-oriented programming in simulation could be considered a collection of knowledged-based systems, in the context of this section we are concerned with interfacing to a knowledge-based system which would conduct higher-level command and control functions. Such a system is being considered for handling intelligence processing to assist the decision making of command and control.[18]

### 4.7.2 Capability

There is no capability built into ROSS itself to interface with other languages or systems. Franzlisp does provide facilities to use foreign

functions. The ability to interface with another module, running alternately or concurrently, would depend on the characteristics of the language and the capability of the operating system.

Interface between the BEM and ANALYST is achieved through inter-process communications by having ROSS write logical symbols into VMS that can be accessed by the LISP environment of the ANALYST. The interface in this case is one-way in that sensor reports are produced by the BEM and read by the ANALYST, that is, there is no flow from the ANALYST back to the BEM.

Interface between the BEM and the Sensor Control Station does operate both ways. The program for the Sensor Control Station is written in ROSS. It provides the Sensor Control Station operator with a display of the terrain, sensor locations and coverages, and locations of reported enemy activity. The operator is provided with a cursor-controlled menu to display sensor data, display sensor reports and change the updating or purging of reports. The operator may also use the menu to initiate a sensor tasking in the BEM. The sensor location and coverage will be indicated at the appropriate time on the graphical displays at both the BEM Control Station and the Sensor Control Station. The sensor reports made by the sensor will be indicated at both stations along with other concurrent events.

## 4.8 Resource Requirements

### 4.8.1 Requirement Elements

The requirement elements to be discussed here include those which are necessary to weigh when considering the use of object-oriented programming

for combat simulation modeling. While the AMIP master plan[12] provides some general guidelines for these elements, no fixed requirements have been determined and must necessarily be subject to tradeoffs for desired levels of breadth, depth and complexity of the simulation.

4.8.2 Requirements

4.8.2.1 Hardware. In order to handle the interpreter, data files and environment for a moderately complex simulation, the minimum computer size is either a mainframe computer (DEC 20) or a 32-bit mini-computer (VAX 11/780). Sufficient text and graphical capability should also be provided to take advantage of the power of object-oriented programming in constructing, following and modifying the simulation. With the increased power provided in the new LISP-processing machines, it is possible that a division-level simulation could be run within acceptable time constraints.

It was found necessary to have 1.25 megabytes of dedicated space to run the BEM with the ROSS environment, interpreter and the size of the BEM simulation. Smaller space was tested but resulted in a large number of page faults and an increase in the running time.

4.8.2.2 Software. Software requirements to use an object-oriented language in a simulation include an appropriate operating system; interpreters and compilers to run the underlying language; an editor; and any requirements peculiar to the particular language (or simulation system). In addition, there are the graphical interfaces to use the graphical equipment in conjunction with the simulation. It was possible to use ROSS to construct the BEM on the VAX 11/780 by obtaining a version of ROSS written in Franzlisp. The EUNICE

emulation package allowed Franzlisp (and ROSS) to be run under the UNIX shell as a job of VMS (the VAX operating system). RAND provided the necessary program code to load and compile the ROSS environment. Software to implement the graphical interface was developed as part of the BEM construction.

4.8.2.3 Personnel Background and Training. The concept of object orientation and message passing is an outgrowth of AI technology. Personnel familiar with AI in general and LISP in particular have little difficulty in learning and applying ROSS. Knowledge of LISP allows the user to take full advantage of ROSS and to develop additional functions as necessary. However, the structure of ROSS makes it quite easy to learn the ROSS commands in a short time even for someone without a LISP background. Nonetheless without understanding LISP, it would be difficult to fully understand ROSS or utilize it to its potential.

4.8.2.4 Set-Up Requirements. The personnel requirements and length of time to construct the simulation model from the ground up depend on many factors such as personnel background, complexity of the simulation, and so forth. Two observations can be made, however, on the initial development of and the changing of one simulation scenario to another.

During development, object-oriented programming allows actors and behaviors to be incrementally created and tested. This provides very quick feedback to the user/analyst and increases understanding of the underlying processes. This understanding can lead to expansion of the model development and can also contribute to the analysis being performed.

After the simulation model has been constructed, changes to incorporate a new scenario can be made in steps to take advantage of insights provided. If the new scenario involves changes in both tactics and terrain, changes to tactics might first be entered to test if differences in results are due to tactics alone. Actor behaviors could be reviewed and changed accordingly, both before and after the test situation.

4.9 Additional Observation

One precaution which must be taken in the use of object-oriented programming is to guard against the misuse of message passing. It is clear that when messages simulate communications between basic objects in the events being simulated, they should be represented in the simulation with messages passed between the objects. However, when there is simply the need to access information in an object's property list, there is a pitfall which can slow the simulation down considerably. ROSS allows the intermixing of ROSS Commands and LISP so that instead of using a ROSS Command such as

(ask <object> recall your <property>)

that information can be accessed with a LISP function of the form

(get <object> <property>)

if the property is stored with the object. (A macro or LISP expression slightly more complex can be written for the job if the property is stored with the objects ancestors.) This gets away from the message pattern matching which can slow the simulation if used too extensively.

The approach we used in implementing BEM in ROSS was to use the message passing where it either represented actual communications between objects or would not subtract from understanding of the simulation. In cases such as performing distance calculations or computing time and space overlaps between units and sensors, we used LISP functions for accessing and operating on data. This allowed us to speed up the simulation as well as to streamline the object behaviors.

## 5.0 SUMMARY AND RECOMMENDATIONS

### 5.1 Summary Findings

Object-oriented programming provides a number of advantages over standard programming in all stages of the construction and use of simulations especially those having to do with combat simulation. The credibility and understanding provided through application of the techniques and use of various languages can serve also to support the credibility of the analysis and studies conducted using such simulation. The English-like code and interactive capability will encourage higher-level users of the analysis being supported to become familiar with the simulation and assist them in assessing the credibility of results provided. The modularity brought about by the storage of data and behaviors with the actors makes it easier to study and modify only the necessary portions of the simulation to bring about desired changes. The dependence of activity upon message passing and actor behaviors simplifies the task of tracing cause and effect within the simulation.

The main limitation of object-oriented programming for use in ground combat simulation appears to be in the running time for a simulation of adequate battle length. The large number of possible interactions between units on both sides taxes the processing capability of any machine when added to the overhead incurred with the features that make object-oriented programming so powerful. The pattern-matching feature of message and behavior handling in particular can produce a heavy processing load when

there are hundreds of units on both sides. While the new LISP machines hold promise to provide better processing capability, it is not known at what additional cost of resources.

The findings presented in the previous section were based on our experience with implementing the BEM in ROSS. The findings cannot necessarily be extended to other object-oriented languages, especially those developed more recently or currently being developed. As more experience is gained in development and application, improvements can be made to overcome shortcomings. It is felt however that, at least with current popular mainframe and mini-computers such as the DEC-20 and the VAX 11/780, the ratio of running-to-CPU time will be a limitation to the use of object-oriented programming for full-scale combat simulation of the level required for the AMIP hierarchy of models. The near term application using current computers in the Army analysis community is in testbed configurations working with two problems that have plagued previous models, those of command and control and intelligence fusion.

## 5.2 Potential Applications

A realizable, practical application offered by object-oriented programming in the near term lies in its use in test-bed configuration. This could either be off-line or as a module tied in with a model of the remainder of the combat or support actions being studied. Tentative behavior rules could be studied in the testbed and then selectively tried in the production model. The testbed could also be used to assist in knowledge-engineering the behavioral rules by providing feedback to the expert on the effects of the rule

or set of rules. As the behavioral rules can represent the decision-making in a simulation, this provides a means of testing and modifying the automated command and control decisions to be implemented in the production model.

Another near-term application to be considered is the use of object-oriented programming for the higher-level interactions while abstracting the lower-level representation and interactions. In a corps level simulation the unit resolution could be at brigade or even division level, depending on the analysis questions to be addressed. The consequent reduction in numbers of units and interactions would substantially reduce running time, but at a cost of detail.

In any simulation application using a technique as unfamiliar as object-oriented programming, it is recommended that one or more levels of model simplifications be developed and maintained. This allows testing of one-sided effects or one-on-one interaction before testing in a full two-sided, many-on-many environment. In the initial simulation development, this is a natural occurrence as the actors and behaviors are incrementally created and tested. Maintenance of stages of the simulation development in archival form help newcomers become familiar with the application of the techniques and the process being simulated.

Due to the processing limitations, a recommendation for full-scale implementation of any AMIP models in object-oriented programming does not appear to be in order at this time. RAND is investigating techniques to overcome the problems by using distributed processing. Another answer may

lie in the use of LISP machines. The increased costs for the additional or more advanced processors may however put the technique out of consideration for use in some simulation applications, at least in the near-term.

## 5.3 Recommendations

It is recommended that a more detailed evaluation of object-oriented programming be supported by the Army. This evaluation should investigate improved efficiencies and developments in ROSS, the balance between the use of object-oriented programming and other programming in the AMIP hierarchy, and the applications of other language-systems (e.g. FLAVORS, LOOPS) incorporating object orientation.

A second recommendation is that research on processing techniques such as that being conducted by Rand be supported by the Army so that current processing limitations can be overcome.

A third recommendation is that the Army should build up an understanding of artificial intelligence, in particular developing techniques such as object-oriented programming in order to get the most use of their application now and in the future.

## GLOSSARY

| | |
|---|---|
| ACTORS | An Object Oriented Language |
| AI | Artificial Intelligence |
| AMIP | Army Model Improvement Program |
| AMMO | AMIP Management Office |
| ANALYST | An expert system for processing intelligence returns |
| BEM | Battlefield Environment Model |
| CAMIS | Command and Management Information System (Laboratory) |
| $C^2$ | Command and Control |
| CM/CB | Counter-mortar/Counter-battery |
| COMINT | Communications Intelligence |
| DIRECTOR | An Object Oriented Language |
| ELINT | Electronic Intelligence |
| EUNICE | A software package allowing UNIX to run as a VMS job |
| FA | Field Artillery |
| FLAVORS | An Object Oriented Language |
| FRANZLISP | A LISP dialect |
| IMINT | Imagery Intelligence |
| INTERLISP | A LISP dialect |
| IPL | Intelligence Processing Laboratory |
| LISP | A symbolic manipulation computer language |
| LOGO | An Object Oriented Language |
| LOOPS | An Object Oriented Language |
| MACLISP | A LISP dialect |
| MTEG | MITRE Threat Event Generator |
| MTI | Moving Target Intelligence |
| PARC | (Xerox's) Palo Alto Research Center |
| ROSS | Rule Oriented Simulation System |
| SCORES | Scenario Oriented Recurring Evaluation System |
| SCIF | Secure Compartmented Information Facility |
| SIMULA | An Object Oriented Language |
| SMALLTALK | An Object Oriented Language |

## GLOSSARY

**(Concluded)**

| | |
|---|---|
| SPL | Secure Processing Laboratory |
| SWIRL | Simulated Warfare in the ROSS Language |
| UNIX | A family of computer operating systems |
| VAX | VAX 11/780 Computer |
| VMS | VAX Operating System |

## REFERENCES


1. The MITRE Corporation, The MITRE Threat Event Generator - Force Movement, MTR-80W00266, Russell P. Bonasso and Emanuel P. Maimone, March 1981.

2. The MITRE Corporation, The MITRE Threat Event Generator (MTEG) - Force Communications, MTR-81W00295, R. P. Bonasso, December 1981.

3. The MITRE Corporation, ANALYST: An Expert System for Processing Sensor Returns, MTP-83W00002, R. P. Bonasso, 1983.

4. The MITRE Corporation, The MITRE Battlefield Environment Model (BEM), MTR-83-?-, R. S. Conker, P. K. Groveston, and R. O. Nugent, 1983.

5. The RAND Corporation, The ROSS Language Manual, N-1884-AF, David McArthur and Philip Klahr, September 1982.

6. John K. Foderaro, Keith L. Sklower, The FRANZLISP Manual, University of California, April 1982.

7. The RAND Corporation, SWIRL: Simulating Warfare in the ROSS Language, N-1885-AF, Philip Klahr, David McArthur, Sanjai Narain, Eric Best, September 1982.

8. George Fisthman, Concepts and Methods in Discrete Event Digital Simulation, Wiley & Sons, 1973.

9. C. Hewitt, "Viewing Control Structures and Patterns of Message Passing", Artificial Intelligence 8 (1977), 323-367.

10. John Markoff, "In Focus: SMALLTALK", Info World, Vol. 5, No. 4, 25-31.

11. K. M. Kahn, Director Guide, AI Memo 482B MIT, 1979.

12. Harold Abelson, "A Beginner's Guide to LOGO", Byte, Aug. 1982, 88-112.

13. Daniel Weinrab and David Moon, LISP Machine Manual, 5th Ed., Cambridge, MA, The MIT Artificial Intelligence Laboratory, Jan. 1983.

14. Mark Steflik, Daniel G. Bobrow, Sanjay Mittal and Lynn Conway, "Knowledge Programming in LOOPS: Report on an Experimental Course", The AI Magazine, Voi. 4, No. 3, Fall 1983, 3-13.

## REFERENCES

## (Concluded)


15. U.S. Army Combined Arms Combat Developments Activity, Standard Scenario for Combat Developments, Europe I, Sequence 2A (U), SECRET.

16. The MITRE Corporation, Threat Definition: Unit Locations and Target Arrays for Soviet Artillery Battalions and Maneuver Battalions, WP-12724, R. R. Darron and D. T. Giles, Jr., 15 November 1977, SECRET.

17. The MITRE Corporation, Threat Definition: Combat Service Support Units, WP-12843, R. R. Darron and D. T. Giles, Jr., 10 February 1978, SECRET.

18. Soviet Ground Force Organizational Guide, Defense Intelligence Agency, June 1978, SECRET NOFORN.

19. AMMO, AMIP Management Plan, 7 Jan. 1983.

20. Discussions, AMMO Office, Mr. Harry Jones, August 1983.

21. Presentation at the National Conference in Artificial Intelligence, August 22-26, 1983, The State of the Art in Computer Learning, Douglas B. Lenat, Stanford University.

22. The MITRE Corporation, Intelligence Fusion Modeling - A Proposed Approach, WP-83W00379, H. James Antonisse, September 1983.

## DISTRIBUTION LIST

### INTERNAL

A-10 C. A. Zraket

D-10 K. E. McVicar

D-14 D. S. Alberts
J. S. McManus
A. J. Tachmindji

W-70 G. Carp
J. Dominitz
E. Famolari
P. G. Freck
F. W. Niedenfuhr
E. L. Rabben
W. A. Tidwell

W-72 C. W. Sanders

W-73 T. H. Nyman

W-74 T. T. Bean
R. P. Bonasso (10)
J. R. Davidson (10)
Z. Z. Friedlander
M. Gale
P. K. Groveston (10)
C. R. Holt
E. P. Maimone
R. O. Nugent (10)
W. E. Zeiner

W70 Information Center

### EXTERNAL

HQDA
DCSOPS DAMO-RQZ
DAMO-RQR
DAMO-RQI

DCSRDA DAMA-RAX-B
DAMA-CSC

ACSI DAMI-ZA
DAMI
DAMI-IS
DAMI-AML

Commander
U.S. Army Electronics Research and Development Command
Director Electronic Warfare Lab
PM-SOTAS
Director Signals Warfare Lab
PM-JTFS

Defense Intelligence Agency

U.S. European Command - EUDAC

Jet Propulsion Laboratories
Mr. S. Freesna
4800 Oak Grove
Pasadena, CA 91109

Institute for Defense Analysis

HQS, TSARCOM PM-SEMA

HQS, AVRADCOM PM-RPV

## DISTRIBUTION LIST

(Continued)

**EXTERNAL (Continued)**

HQS, CERCOM

HQS, EMRA

The Army Model Management Office
U.S. Army Combined Arms Center
ATTN: ATZL-CAN-DO
Mr. Harry Jones (20)
Ft. Leavenworth, KS 66027

Army Library
ATTN: ANR-AL-RS
(Army Studies)
Room 1A518
Pentagon
Washington D.C. 20310

Commander
Defense Technical Information Center
ATTN: DDA
Cameron Station
Alexandria, VA 22314 (2)

Commandant
U.S. Army Command and General Staff College
Ft. Leavenworth, KS 66027

Commandant
U.S. Army War College

HQ DARCOM
DRCBSI
5001 Eisenhower Avenue
Alexandria, VA 22333

Commander
U.S. Army Combined Arms Center
ATZL-CSC-I (LTC Schneider,
Mr. Kroening, CPT Linn)
ATZL-CAS-W (LTC Cookingham)
ATZL-TAC-LO (LTC Bridger)
Ft. Leavenworth, KS 66027

Commander
U.S. Army Nuclear and Chemical Agency
ATTN: MONA-OPS
Ft. Belvoir, VA 22060 (4)

Commander
U.S. Army Infantry Center
ATTN: ATSH-CD-CSO-OR
(COL Skaife)
Ft. Benning, GA 31905

Commander
U.S. Army Aviation Center
ATTN: ATZQ-D-CS
(COL Burnett)
Ft. Rucker, AL

Commander
U.S. Army Air Defense School
ATTN: ATSA-CDS-F
Ft. Bliss, TX 79916

Commander
U.S. Army Logistics Center
ATTN: ATCL-O
(COL Young)
Ft. Lee, VA 23801

## DISTRIBUTION LIST

(Continued)

**EXTERNAL (Continued)**

Commander
U.S. Army Field Artillery School
ATTN: ATSF-CA
(MAJ Henry)
Ft. Sill, OK 73503

Commander
U.S. Army Armor Center
ATTN: ATZK-CD-SD
(MAJ Shepard)
Ft. Knox, KY 40121

Commander
U.S. Army Intelligence Center
ATTN: ATSI-CD-CS
(LTC Daniel)
Ft. Huachuca, AZ 85613

Commander
Soldier Support Center
ATTN: ATSG-DCD-AD
(LTC Crosby)
Ft. Benjamin Harrison, IN 46216

HQ, Department of the Army
SAUS-OR (Mr. Hollis, Dr. Fallin)
DAMO-ZD (Mr. Vandiver)
DAMA-ZD (Mr. Woodall)
DAMI-FRT (Mr. Beuch)
DACS-DMO (Ms. Langston)
DALO-PLF (COL Wakefield)
DAAC-PE (COL Sievert)
DAPE-PSS (LTC Herrick)
DAMI-RQT (COL Kressler)
DASG-HCD (LTC Arnt)
DAMO-CP4 (COL Greenwood)
ASA (IL&FM) (Mr. Rosenblum)
Washington D.C. 20310

Director, U.S. Army Concepts
Analysis Agency
ATTN: CSCA-AZ (Mr. Hardison)
CSCA-MC (Mr. Louer,
Mr. Shedlowsky,
LTC Deems) (6)
8120 Woodmont Avenue
Bethesda, MD 20014

Director, U.S. Army Materiel Systems
Analysis Activity
ATTN: DRXSY-C (Mr. Myers)
DRXSY-GR (Mr. Clifford)
DRXSY-GS (Mr. Brooks)
DRXSY-FM (Mr. Blanton) (4)
Aberdeen Proving Ground, MD 21005

Commander
Director, U.S. Army TRADOC Systems
Analysis Activity
ATTN: ATAA-D (Mr. Goode)
ATAA-TG (Mr. Carrillo) (4)
ATAA-TC (Mr. Matheson)
ATAA-TCF (Mr. McCoy)
White Sands, NM 88002

Director, U.S. Army Research Institute
ATTN: PERI-SZ (Dr. Johnson)
5001 Eisenhower Avenue
Alexandria, VA 22333

## DISTRIBUTION LIST

(Concluded)

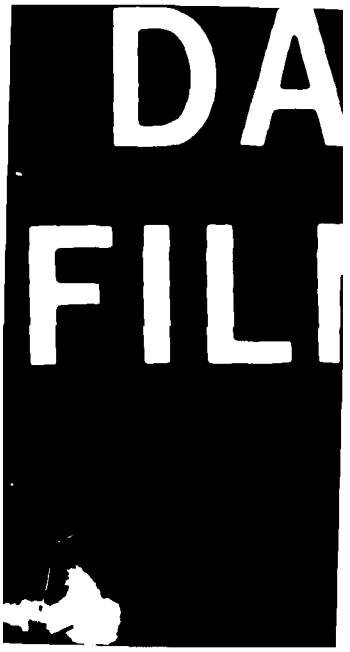
**EXTERNAL (Concluded)**

Dr. Wilbur Payne
Director TRADOC Operations Research
Activity
White Sands, NM 88002

Deputy Commander, Combined Arms
Operations Research Activity
ATTN: ATOR-CAA-DC (COL West)
ATOR-CAA-DR (Mr. Pleger) (10)
ATOR-CAT-D (LTC Childs) (3)
Ft. Leavenworth, KS 66027

Commander
U.S. Army Intelligence and Threat
Analysis Center
ATTN: IAX-I (COL Satterwaite)
IAX-I-OR (Mr. Carroad) (5)
Arlington Hall Station, VA 22022

Commander
U.S. Army Training and Doctrine
Command
ATTN: ATCG-S (Mr. Christman)
ATCD-D (COL Shoffner)
ATCD-AT (Mr. Goldberg)
ATDO (COL Lundgren)
Ft. Monroe, VA 23651


DA
FIL